Web Dev Day 6: JavaScript Guide Part 1

Bhupesh Kumar
Posted on Feb 9 • Edited on Feb 28

#webdev #programming #javascript #coding

What is a Variable?

A variable in JavaScript is a named container that holds a value. It allows us to store, update, and retrieve data dynamically.

JavaScript provides three ways to declare variables:

- `var` (old syntax)
- `let` (modern)
- `const` (for constants)

```
let name = "Bhupesh";
const age = 25;
var country = "Canada";
```

Data Types in JS

JavaScript has **primitive** and **non-primitive** data types.

Primitive Data Types

These are immutable and stored directly in memory:

- **String** → `"Hello"`
- **Number** → `42`
- **Boolean** → `true`, `false`
- **Undefined** → `let x;` (not assigned)
- **Null** → `let y = null;` (empty value)
- **BigInt** → `1234567890123456789012345678901234567890123n`
- **Symbol** → `Symbol('unique')`

Non-Primitive Data Types

These are reference types and stored as objects:

- **Objects** → `{ name: "John", age: 30 }`
- **Arrays** → `["apple", "banana", "cherry"]`
- **Functions** →

```
function greet() {
    console.log("Hello!");
}
```

Numbers in JavaScript

JavaScript has only **one type** for numbers: **floating-point numbers**.

Example:

```
let a = 10;       // Integer
let b = 10.5;     // Float
let c = 1e3;      // Scientific notation (1000)
```

Special Number Values:

- Infinity → `console.log(1 / 0);`
- -Infinity → `console.log(-1 / 0);`
- NaN (Not-a-Number) → `console.log("hello" * 2);`

Operations in JavaScript

JavaScript supports the following **operators**:

- **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%`, `**`
- **Comparison Operators**: `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=`
- **Logical Operators**: `&&`, `||`, `!`
- **Bitwise Operators**: `&`, `|`, `^`, `~`, `<<`, `>>`
- **Ternary Operator**: `condition ? trueValue : falseValue`

Example:

```
console.log(5 + 2);  // 7
console.log(10 / 2); // 5
console.log(3 ** 2); // 9 (Exponentiation)
```

NaN in JavaScript

`NaN` stands for **"Not-a-Number."** It occurs when an operation does not yield a valid number.

Example:

```
console.log("hello" * 2); // NaN
console.log(0 / 0);       // NaN
```

Operator Precedence in JavaScript

Operator precedence determines how expressions are evaluated in JavaScript.

Operator Type	Operators	Precedence
Parentheses	`()`	**Highest**
Exponentiation	`**`	2
Multiplication, Division, Modulus	`*`, `/`, `%`	3
Addition, Subtraction	`+`, `-`	4
Relational	`<`, `>`, `<=`, `>=`	5
Equality	`==`, `===`, `!=`, `!==`	6
Logical AND	`&&`	7
Logical OR	`	

Example:

{% raw %}

```
console.log(5 + 3 * 2);    // 11 (Multiplication first)
console.log((5 + 3) * 2);  // 16 (Parentheses first)
```

`let` Keyword

The `let` keyword was **introduced in ES6**. It has the following properties:

- **Block-scoped** (Only accessible within the block `{}` it is declared in).
- **Can be reassigned** but **cannot be redeclared** within the same scope.

Example:

```
let x = 10;
x = 20;  // ✔️ Allowed

let x = 30; // ❌ Error (Cannot redeclare)
```

`const` Keyword

The `const` keyword has the following properties:

- **Block-scoped** (Like `let`, it is confined within the block `{}` it is declared in).
- **Cannot be reassigned** after declaration.
- **Cannot be redeclared** in the same scope.
- **Must be initialized** when declared.

Example:

```
const PI = 3.1416;
PI = 3.14; // ❌ Error (Cannot reassign)
```

`var` Keyword (Old Syntax)

The `var` keyword is **function-scoped**, meaning:

- **Can be redeclared** and **reassigned**.
- **Not recommended** in modern JavaScript due to scoping issues.

Example:

```
var name = "Alice";
var name = "Bob";  // ✔️ Allowed (but not good practice)
```

Assignment Operators in JavaScript

Assignment operators are used to **assign values to variables**.

Operator	Example	Meaning
`=`	`x = 5`	Assigns `5` to `x`

Operator	Example	Meaning
`+=`	`x += 2`	`x = x + 2`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 4`	`x = x * 4`
`/=`	`x /= 5`	`x = x / 5`
`%=`	`x %= 2`	`x = x % 2`

Unary Operators in JavaScript

Unary operators operate on a **single operand**.

Operator	Description	Example	Output
`+` (Unary plus)	Converts value to a number	`+"42"`	`42`
`-` (Unary minus)	Negates a number	`-10`	`-10`
`++` (Increment)	Increases value by `1`	`let x = 1; x++`	`2`
`--` (Decrement)	Decreases value by `1`	`let y = 2; y--`	`1`
`typeof`	Returns the type of a value	`typeof 42`	`"number"`

Identifiers Rule in JavaScript

Identifiers are **names used for variables, functions, and objects**. JavaScript follows these rules:

Valid Identifiers:

- Can include **letters, digits, underscores (`_`)**, and **dollar signs (`$`)**.
- **Cannot start with a digit**.
- **JavaScript is case-sensitive**.

Valid Identifiers:

```
let firstName;
let _privateVar;
let $price;
```

Invalid Identifiers in JavaScript

JavaScript has strict rules for naming **identifiers**. The following are **invalid identifiers**:

```
let 2name;    // ❌ Error (Cannot start with a digit)
let my-name;  // ❌ Error (Hyphens are not allowed)
```

camelCase Naming Convention in JavaScript

JavaScript follows the **camelCase** naming convention, where:

- The first word is lowercase.
- Each subsequent word starts with an uppercase letter.

Example:

```
let userName;
let totalAmount;
let firstName;
```

Boolean in JavaScript

A **Boolean** represents either `true` or `false`.

Example:

```
let isLoggedIn = true;
let hasDiscount = false;
```

Boolean Conversion:

```
console.log(Boolean(0));        // false
console.log(Boolean("Hello"));  // true
```

What is TypeScript?

TypeScript is a **superset of JavaScript** that:

- **Adds static typing** to JavaScript.
- **Prevents runtime errors** by catching issues during development.
- **Compiles to JavaScript**, making it compatible with all JS environments.

Static vs. Dynamic Typing:

- **TypeScript** is **statically typed**, meaning variable types are checked at compile time.
- **JavaScript** is **dynamically typed**, meaning types are inferred at runtime.

Example:

```
let age: number = 25;  // TypeScript
```

String in JavaScript

A **string** is a sequence of characters enclosed in **quotes (`""` or `''`)**.

Example:

```
let greeting = "Hello, World!";
console.log(greeting.length); // 13
```

String Indices in JavaScript

Each character in a **string** has an **index**, starting from `0`.

Example:

```
let str = "JavaScript";
console.log(str[0]);              // "J"
console.log(str[str.length - 1]); // "t"
```

String Concatenation in JavaScript

Concatenation is the process of **joining two or more strings together**.

Methods of Concatenation:

1. **Using the `+` operator** (Most common)
2. **Using the `.concat()` method** (Less common)

Example:

```
console.log("Hello" + " " + "World"); // "Hello World"
console.log("Hi".concat(" there!"));  // "Hi there!"
```

`null` and `undefined` in JavaScript

In JavaScript, `null` and `undefined` are **special values** that represent **the absence of a value**, but they have different meanings.

`undefined`

- A variable is `undefined` when it is **declared but not assigned a value**.
- It is the **default value** of an uninitialized variable.

Example:

```
let x;
console.log(x); // undefined
```

`null` in JavaScript

In JavaScript, `null` represents an **intentional absence of any object value**.

- It signifies that a variable **intentionally holds no value**.
- `null` must be **explicitly assigned**.

Example:

```
let y = null;
console.log(y); // null
```

`console.log()` in JavaScript

`console.log()` is a built-in JavaScript method used to print **output to the console**.

Usage:

```
console.log("Hello, World!"); // Output: Hello, World!
console.log(5 + 3);           // Output: 8
```

Linking a JavaScript File to HTML

To use JavaScript in an HTML file, you need to **link an external JavaScript file** using the `<script>` tag.

How to Link an External JavaScript File

1. **Create an HTML file (`index.html`)**
2. **Create a JavaScript file (`script.js`)**
3. **Link the JavaScript file in the HTML file**

Example:

`index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JavaScript Linking</title>
</head>
<body>
    <h1>Welcome to JavaScript</h1>

    <!-- Linking JavaScript -->
    <script src="script.js"></script>
</body>
</html>
```

Template Literals in JavaScript

Template literals (**also called template strings**) allow for **easier string manipulation** using **backticks ()** instead of quotes.

Syntax:

```
This is a template literal
```

String Interpolation

Use ${} to insert variables or expressions directly into a string.

```
let name = "Alice";
let age = 25;

console.log(`My name is ${name} and I am ${age} years old.`);
// Output: My name is Alice and I am 25 years old.
```

Multi-line Strings

Template literals allow multi-line strings without needing \n.

```
let message = `This is line 1
This is line 2
This is line 3`;

console.log(message);
```

Expression Evaluation

You can use expressions inside ${}.

```
let a = 5, b = 10;

console.log(`The sum of ${a} and ${b} is ${a + b}.`);
// Output: The sum of 5 and 10 is 15.
```

Operators in JavaScript

JavaScript supports various **operators** to perform operations on variables and values.

Arithmetic Operators

Used to perform mathematical operations.

Operator	Description	Example	Output
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`10 - 4`	`6`
`*`	Multiplication	`2 * 5`	`10`
`/`	Division	`10 / 2`	`5`
`%`	Modulus (Remainder)	`10 % 3`	`1`
`**`	Exponentiation (Power)	`3 ** 2`	`9`

Example:

```
console.log(5 + 2);  // 7
console.log(10 / 2); // 5
console.log(3 ** 2); // 9 (Exponentiation)
```

Assignment Operators in JavaScript

Assignment operators are used to **assign values to variables**.

List of Assignment Operators

Operator	Example	Meaning
`=`	`x = 5`	Assigns `5` to `x`
`+=`	`x += 2`	`x = x + 2`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 4`	`x = x * 4`
`/=`	`x /= 5`	`x = x / 5`
`%=`	`x %= 2`	`x = x % 2`

🔹 Example Usage

```
let x = 10;
x += 5;   // x = 15
x *= 2;   // x = 30
console.log(x); // Output: 30
```

Comparison Operators in JavaScript

Comparison operators are used to **compare values** and return a Boolean (`true` or `false`).

List of Comparison Operators

Operator	Description	Example	Output
`==`	Equal to (loose comparison)	`5 == "5"`	`true`
`===`	Strictly equal (Type + Value)	`5 === "5"`	`false`
`!=`	Not equal	`5 != "5"`	`false`
`!==`	Strictly not equal	`5 !== "5"`	`true`
`>`	Greater than	`10 > 5`	`true`
`<`	Less than	`3 < 5`	`true`
`>=`	Greater than or equal to	`10 >= 10`	`true`
`<=`	Less than or equal to	`3 <= 5`	`true`

Example Usage

```
console.log(5 == "5");  // true (loose comparison)
console.log(5 === "5"); // false (strict comparison)
```

`5 == '5'` in JavaScript

In JavaScript, the expression `5 == '5'` evaluates to **`true`**.

This happens because `==` is the **loose equality operator**, which **performs type coercion** before comparing values.

Loose Equality (`==`) vs. Strict Equality (`===`)

Expression	Result	Explanation
`5 == '5'`	`true`	Loose equality (`==`) **converts types** before comparison. The string `'5'` is converted to a number `5`.
`5 === '5'`	`false`	Strict equality (`===`) **does not perform type conversion**. Since `5` (number) and `'5'` (string) have different types, the result is `false`.

Example Usage

```
console.log(5 == '5');  // true (loose comparison with type coercion)
console.log(5 === '5'); // false (strict comparison without type coercion)
```

Logical Operators in JavaScript

Logical operators are used to **combine conditions** and return a Boolean (`true` or `false`).

List of Logical Operators

Operator	Description	Example	Output
`&&`	Logical AND	`true && false`	`false`
`		`	`	Logical OR
`!`	Logical NOT	`!true`	`false`

Example Usage

```
console.log(true && false);  // false
console.log(true || false);  // true
console.log(!true);          // false
```

Bitwise Operators in JavaScript

Bitwise operators are used to perform **binary calculations** by manipulating bits directly.

List of Bitwise Operators

Operator	Description	Example
`&`	AND	`5 & 1`

Operator	Description	Example
'	'	OR
^	XOR	5 ^ 1
~	NOT	~5
<<	Left shift	5 << 1
>>	Right shift	5 >> 1

Example Usage

```
console.log(5 & 1);  // 1  (AND operation)
console.log(5 | 1);  // 5  (OR operation)
console.log(5 ^ 1);  // 4  (XOR operation)
console.log(~5);     // -6 (NOT operation)
console.log(5 << 1); // 10 (Left shift)
console.log(5 >> 1); // 2  (Right shift)
```

Ternary Operator in JavaScript

The **ternary operator** is a shorthand for `if...else` statements.

It allows writing conditional logic in a **compact** and **readable** way.

Syntax:

```
condition ? trueValue : falseValue;
```

Comparisons for Non-Numbers in JavaScript

JavaScript allows **comparisons** not only for numbers but also for **strings, Booleans, and other data types**.

When comparing **non-numeric values**, JavaScript applies **type conversion rules**.

Comparing Strings

- JavaScript compares **strings lexicographically** (alphabetical order) based on **Unicode values**.
- Uppercase letters (`A-Z`) come **before** lowercase letters (`a-z`).

Example:

```
console.log("apple" > "banana");  // false (because "a" comes before "b")
console.log("Zoo" > "apple");     // false (uppercase "Z" comes before lowercase
"a")
console.log("Hello" > "hello");   // false (uppercase "H" < lowercase "h")
```

Comparing Booleans in JavaScript

In JavaScript, **Booleans (`true` and `false`) are converted to numbers** when used in comparisons:

- `false` is treated as `0`
- `true` is treated as `1`

Example:

```
console.log(true > false);  // true (1 > 0)
console.log(false == 0);    // true (false → 0)
console.log(true == 1);     // true (true → 1)
console.log(false < true);  // true (0 < 1)
```

Comparing `null` and `undefined` in JavaScript

JavaScript treats `null` and `undefined` **differently** in comparisons.

Example:

```
console.log(null == undefined);  // true (special case)
console.log(null === undefined); // false (strict comparison)
console.log(null > 0);           // false
console.log(null == 0);          // false
console.log(null >= 0);          // true (strange behavior)
console.log(undefined > 0);      // false
console.log(undefined == 0);     // false
console.log(undefined >= 0);     // false
```

Comparing Objects in JavaScript

In JavaScript, **objects are not directly comparable**.

They are compared by **reference**, not by value.

Example:

```
let obj1 = { name: "Alice" };
let obj2 = { name: "Alice" };

console.log(obj1 == obj2);  // false (different memory references)
console.log(obj1 === obj2); // false (even though they look the same)
```

Conditional Statements in JavaScript

Conditional statements **allow a program to make decisions** based on conditions.

JavaScript provides multiple ways to handle conditions.

`if` Statement

The `if` statement executes code **only if a condition is true**.

Syntax:

```
if (condition) {
    // Code to execute if the condition is true
}
```

`if...else` Statement in JavaScript

The **`if...else` statement** executes one block of code **if the condition is true** and another block **if it is false**.

Example:

```
let age = 16;

if (age >= 18) {
    console.log("You can vote.");
} else {
    console.log("You cannot vote.");
}
// Output: "You cannot vote."
```

`if...else if...else` Statement in JavaScript

The **`if...else if...else` statement** is used when multiple conditions **need to be checked sequentially**.

JavaScript **evaluates each condition in order**, and the first condition that evaluates to `true` gets executed.

Example:

```
let score = 85;

if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 80) {
    console.log("Grade: B");
} else if (score >= 70) {
    console.log("Grade: C");
} else {
    console.log("Grade: F");
}
// Output: "Grade: B"
```

Ternary Operator (`? :`) in JavaScript

The **ternary operator** is a **shorter** and **more concise** way to write an `if...else` statement.

It is useful for simple conditions where you **want to assign a value or execute a statement** based on a condition.

Syntax:

```
condition ? trueValue : falseValue;
```

`switch` Statement in JavaScript

The `switch` **statement** is used when a variable **needs to be compared against multiple values**.

It is often used as a cleaner alternative to multiple `if...else if...else` statements.

Syntax:

```
switch (expression) {
    case value1:
        // Code to execute if expression === value1
        break;
    case value2:
        // Code to execute if expression === value2
        break;
    default:
        // Code to execute if no cases match
}
```

Nested `if...else` in JavaScript

A **nested `if...else` statement** is an `if...else` inside another `if` or `else`.

It allows checking **multiple conditions** in a hierarchical manner.

Syntax:

```
if (condition1) {
    if (condition2) {
        // Code to execute if both conditions are true
    } else {
        // Code to execute if condition1 is true but condition2 is false
    }
} else {
    // Code to execute if condition1 is false
}
```

Logical Operators in JavaScript

Logical operators are used to **combine multiple conditions** and return a Boolean (`true` or `false`).

They are essential for making **decisions** in conditional statements.

List of Logical Operators

Operator	Description	Example	Output
`&&`	Logical AND	`true && false`	`false`
`	`	`	`	Logical OR
`!`	Logical NOT	`!true`	`false`

Example Usage

```
console.log(true && false);  // false
console.log(true || false);  // true
console.log(!true);          // false
```

Truthy and Falsy Values in JavaScript

In JavaScript, **every value is either "truthy" or "falsy"** when evaluated in a Boolean context.

- **Truthy values** behave like `true` when used in a condition.

- **Falsy values** behave like `false` when used in a condition.

Falsy Values

A **falsy** value is a value that evaluates to `false` in a Boolean context.

JavaScript has **exactly 8 falsy values**:

Falsy Value	Description
`false`	Boolean `false` itself
`0`	Number zero
`-0`	Negative zero
`0n`	BigInt zero
`""`	Empty string
`null`	Absence of a value
`undefined`	Uninitialized variable
`NaN`	Not-a-Number

Example:

```
if (0) {
    console.log("Truthy");
} else {
    console.log("Falsy");
}
// Output: "Falsy"

if ("") {
    console.log("Truthy");
} else {
    console.log("Falsy");
}
// Output: "Falsy"
```

Truthy Values in JavaScript

A **truthy value** is **any value that is not falsy**.

In JavaScript, **all values are truthy unless they are in the list of falsy values** (`false`, `0`, `""`, `null`, `undefined`, `NaN`, etc.).

Common Truthy Values

Truthy Value	Example
Non-empty strings	`"hello"`, `"false"`, `"0"`
Numbers except `0`	`42`, `-1`, `3.14`
Non-empty arrays	`[]`
Non-empty objects	`{}`
Functions	`function() {}`
Special values	`" "` (space), `"0"` (string zero)

Example:

```
if ("Hello") {
    console.log("Truthy");
} else {
    console.log("Falsy");
}
// Output: "Truthy"

if (42) {
    console.log("Truthy");
} else {
    console.log("Falsy");
}
// Output: "Truthy"
```

Alerts and Prompts in JavaScript

JavaScript provides built-in methods to interact with users using **alerts, prompts, and confirmations**.

`alert()`

The `alert()` method **displays a pop-up message** to the user.

Syntax:

```
alert("This is an alert!");
```

`prompt()` in JavaScript

The **`prompt()` method** is used to **ask the user for input** via a pop-up dialog.

It returns the user input **as a string**, or `null` if the user clicks **"Cancel"**.

Syntax:

```
let userInput = prompt("Enter your name:");
```

`confirm()` in JavaScript

The `confirm()` **method** displays a pop-up **asking for user confirmation**.

It returns:

- `true` if the user clicks **"OK"**.
- `false` if the user clicks **"Cancel"**.

Syntax:

```
let result = confirm("Are you sure?");
```

String Methods in JavaScript

JavaScript provides many **built-in methods** to manipulate and work with strings.

These methods help in **searching, replacing, slicing, and modifying strings** efficiently.

Common String Methods

Method	Description	Example
`.length`	Returns the string length	`"Hello".length` → `5`
`.toUpperCase()`	Converts to **uppercase**	`"hello".toUpperCase()` → `"HELLO"`
`.toLowerCase()`	Converts to **lowercase**	`"HELLO".toLowerCase()` → `"hello"`
`.charAt(index)`	Returns character at **index**	`"JavaScript".charAt(4)` → `"S"`
`.indexOf("str")`	Returns **first occurrence index**	`"hello".indexOf("l")` → `2`
`.lastIndexOf("str")`	Returns **last occurrence index**	`"hello".lastIndexOf("l")` → `3`
`.includes("str")`	Checks if string contains **substring**	`"hello".includes("he")` → `true`
`.startsWith("str")`	Checks if string **starts with** substring	`"hello".startsWith("he")` → `true`
`.endsWith("str")`	Checks if string **ends with** substring	`"hello".endsWith("lo")` → `true`
`.slice(start, end)`	Extracts substring **(end not included)**	`"JavaScript".slice(0, 4)` → `"Java"`

Method	Description	Example
`.substring(start, end)`	Similar to `.slice()`, but can't accept negative indexes	`"JavaScript".substring(0, 4)` → `"Java"`
`.substr(start, length)`	Extracts substring of specific **length**	`"JavaScript".substr(4, 6)` → `"Script"`
`.replace("old", "new")`	Replaces first match	`"hello world".replace("world", "JS")` → `"hello JS"`
`.replaceAll("old", "new")`	Replaces **all occurrences**	`"apple apple".replaceAll("apple", "banana")` → `"banana banana"`
`.split("delimiter")`	Splits into an **array**	`"one,two,three".split(",")` → `["one", "two", "three"]`
`.trim()`	Removes spaces from **start & end**	`" hello ".trim()` → `"hello"`
`.repeat(n)`	Repeats string **n times**	`"Hi ".repeat(3)` → `"Hi Hi Hi "`

Example Usage:

```
let text = "JavaScript is fun!";

console.log(text.length);          // 18
console.log(text.toUpperCase());   // "JAVASCRIPT IS FUN!"
console.log(text.toLowerCase());   // "javascript is fun!"
console.log(text.charAt(0));       // "J"
console.log(text.includes("fun")); // true
console.log(text.startsWith("Java")); // true
console.log(text.endsWith("fun!"));  // true
console.log(text.slice(0, 10));     // "JavaScript"
console.log(text.replace("fun", "awesome")); // "JavaScript is awesome!"
console.log(text.split(" "));      // ["JavaScript", "is", "fun!"]
console.log("  hello  ".trim());   // "hello"
```

`trim()` Method in JavaScript

The **`trim()` method** removes **whitespace** from both **ends** of a string (**start & end**), without affecting the spaces **inside** the string.

Syntax:

```
string.trim();
```

Strings are Immutable in JavaScript

In JavaScript, **strings are immutable**, meaning **they cannot be changed after they are created**.

Any operation that appears to modify a string **actually returns a new string**, leaving the original string unchanged.

Example:

```
let str = "Hello";
str[0] = "J";  // Attempt to change the first letter
console.log(str);
// Output: "Hello" (unchanged)
```

String Methods with Arguments in JavaScript

Many **string methods** in JavaScript accept **arguments**, allowing you to manipulate strings dynamically.

Common String Methods with Arguments

Method	Description	Example
`.charAt(index)`	Returns the character at a specific **index**	`"JavaScript".charAt(4)` → `"S"`
`.slice(start, end)`	Extracts a substring **(end not included)**	`"JavaScript".slice(0, 4)` → `"Java"`
`.substring(start, end)`	Extracts substring **(similar to `.slice()`)**	`"JavaScript".substring(4, 10)` → `"Script"`
`.substr(start, length)`	Extracts a substring of **specific length**	`"JavaScript".substr(4, 6)` → `"Script"`
`.replace("old", "new")`	Replaces the **first occurrence**	`"hello world".replace("world", "JS")` → `"hello JS"`
`.replaceAll("old", "new")`	Replaces **all occurrences**	`"apple apple".replaceAll("apple", "banana")` → `"banana banana"`
`.split("delimiter")`	Splits into an **array** based on a delimiter	`"one,two,three".split(",")` → `["one", "two", "three"]`
`.repeat(n)`	Repeats the string **n times**	`"Hi ".repeat(3)` → `"Hi Hi Hi "`

`indexOf()` Method in JavaScript

The **`indexOf()` method** returns the **first occurrence index** of a specified substring within a string.

If the substring **is not found**, it returns `-1`.

Syntax:

```
string.indexOf(searchValue, startIndex);
```

Method Chaining in JavaScript

Method chaining is a technique in JavaScript where **multiple methods** are called on an object **in a single statement**.

This makes the code **cleaner and more readable**.

Example:

```
let text = "  JavaScript Method Chaining  ";

let result = text.trim().toUpperCase().replace("CHAINING", "MAGIC");

console.log(result);
// Output: "JAVASCRIPT METHOD MAGIC"
```

`slice()` Method in JavaScript

The **`slice()` method** extracts a portion of a string and returns **a new string without modifying** the original string.

Syntax:

```
string.slice(startIndex, endIndex);
```

`replace()` Method

The **`replace()` method** searches for a specified substring and replaces it with a new value.

By default, it **only replaces the first occurrence** unless you use `replaceAll()`.

Syntax:

```
string.replace(searchValue, newValue);
```

`repeat()` Method in JavaScript

The `repeat()` **method** creates a **new string** by repeating the original string `n` **times**.

Syntax:

```
string.repeat(n);
```

Array Data Structure in JavaScript

An **array** in JavaScript is a data structure that allows **storing multiple values** in a **single variable**.

It can hold elements of **different data types**, including numbers, strings, objects, and even other arrays.

Creating an Array

Using Square Brackets `[]` (Most Common)

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits);
// Output: ["Apple", "Banana", "Cherry"]
```

Creating an Array Using `new Array()` in JavaScript

JavaScript allows you to create arrays using the `new Array()` constructor.

This method is **less commonly used** than square brackets `[]`, but it's useful in certain cases.

Syntax:

```
let arrayName = new Array(element1, element2, ...);
```

Mixed Arrays in JavaScript

A **mixed array** in JavaScript is an array that contains **multiple data types**, including:

- Numbers
- Strings
- Booleans
- Objects
- Arrays
- Functions

Creating a Mixed Array

JavaScript **allows arrays to hold multiple data types** in the same array.

```
let mixedArray = [42, "Hello", true, { name: "Alice" }, [1, 2, 3], function() {
return "Hi!"; }];

console.log(mixedArray);
// Output: [42, "Hello", true, {name: "Alice"}, [1, 2, 3], function]
```

Accessing Characters in a String Inside an Array

In JavaScript, when you store a **string inside an array**, you can **access individual characters** using **indexing**.

Example:

```
let arr = ["bhupesh", 1, 2];

console.log(arr[0][0]); // "b"
console.log(arr[0][1]); // "h"
console.log(arr[0][2]); // "u"
```

Arrays are Mutable in JavaScript

In JavaScript, **arrays are mutable**, meaning their elements **can be modified, added, or removed** without changing their reference in memory.

Example: Modifying an Array

```
let numbers = [1, 2, 3, 4];

numbers[1] = 99;  // Changing the value at index 1

console.log(numbers);
// Output: [1, 99, 3, 4]
```

Array Methods in JavaScript

JavaScript provides various **built-in array methods** to manipulate and process arrays efficiently.

Common Array Methods

Method	Description	Example
`.push(value)`	Adds an element **to the end**	`arr.push("Mango")`
`.pop()`	Removes **the last element**	`arr.pop()`

Method	Description	Example
`.unshift(value)`	Adds an element **to the beginning**	`arr.unshift("First")`
`.shift()`	Removes **the first element**	`arr.shift()`
`.indexOf(value)`	Returns **index of value**	`arr.indexOf("Apple")`
`.includes(value)`	Checks if array **contains value**	`arr.includes("Banana")` → `true`
`.slice(start, end)`	Returns a **portion of an array**	`arr.slice(1, 3)`
`.splice(start, deleteCount, item)`	Removes or adds elements	`arr.splice(2, 1, "New")`
`.reverse()`	Reverses the array	`arr.reverse()`
`.sort()`	Sorts array **alphabetically**	`arr.sort()`
`.concat(arr2)`	Combines two arrays	`arr.concat(arr2)`
`.join(separator)`	Converts array **to string**	`arr.join(", ")`
`.map(fn)`	Creates a **new array** by applying function	`arr.map(x => x * 2)`
`.filter(fn)`	Returns elements that match a **condition**	`arr.filter(x => x > 5)`
`.reduce(fn, initialValue)`	Reduces array to **a single value**	`arr.reduce((sum, num) => sum + num, 0)`
`.forEach(fn)`	Iterates over each element	`arr.forEach(x => console.log(x))`

Example Usage:

Adding & Removing Elements

```
let fruits = ["Apple", "Banana", "Cherry"];

fruits.push("Mango");  // Adds to the end
fruits.unshift("Mango"); // Adds to the beginning

console.log(fruits);
// Output: ["Mango", "Apple", "Banana", "Cherry"]
```

Extracting a Portion of an Array (`slice()`)

The `slice()` **method** extracts a portion of an array **without modifying** the original array.

Example:

```
let colors = ["Red", "Green", "Blue", "Yellow"];

let slicedColors = colors.slice(1, 3);
console.log(slicedColors);
// Output: ["Green", "Blue"]
```

Modifying an Array Using `splice()` in JavaScript

The `splice()` **method** allows you to **add, remove, or replace** elements in an array by modifying the original array.

Syntax:

```
array.splice(startIndex, deleteCount, item1, item2, ...);
```

Transforming an Array Using `map()` in JavaScript

The `map()` **method** creates a **new array** by applying a function to **each element** of an existing array.

It **does not modify** the original array.

Syntax:

```
array.map(function(element, index, array) {
    return modifiedElement;
});
```

Filtering an Array (`filter()`)

The `filter()` **method** returns a **new array** containing only elements that match a specified condition.

Syntax:

```
array.filter(function(element, index, array) {
    return condition;
});
```

Reducing an Array Using `reduce()` in JavaScript

The `reduce()` **method** reduces an array to **a single value** by applying a function.

It is commonly used for **summation, finding maximum/minimum values, and more**.

Syntax:

```
array.reduce(function(accumulator, element, index, array) {
    return newAccumulator;
}, initialValue);
```

Issue with Sorting Numbers Using `sort()` in JavaScript

By default, the `sort()` **method** in JavaScript **treats numbers as strings**,

leading to **incorrect sorting results**.

Incorrect Sorting (Default Behavior)

Example:

```
let numbers = [10, 5, 40, 25, 1];

console.log(numbers.sort());
// Output: [1, 10, 25, 40, 5] (Incorrect)
```

- The sort() method converts numbers to strings before sorting.
- When sorting strings, JavaScript compares character by character (lexicographical order).
- Since "1" comes before "5", "10" is placed before "5".

Array References in JavaScript

In JavaScript, **arrays are reference types**, meaning they are **stored by reference** rather than by value.

This affects how arrays are assigned, compared, and modified.

Example: Arrays Are Stored by Reference

```
let arr1 = [1, 2, 3];
let arr2 = arr1;  // Both variables point to the same array

arr2.push(4);

console.log(arr1); // Output: [1, 2, 3, 4]
console.log(arr2); // Output: [1, 2, 3, 4]
```

Constant Arrays in JavaScript (`const` Arrays)

In JavaScript, **arrays declared with `const` are not immutable**.

You **cannot reassign** a `const` array, but you **can modify its contents**.

Declaring a Constant Array

```
const numbers = [1, 2, 3, 4];

console.log(numbers);
// Output: [1, 2, 3, 4]
```

Nested Arrays in JavaScript (Multi-Dimensional Arrays)

A **nested array** (or **multi-dimensional array**) is an array that **contains other arrays** as elements.

These are useful for representing **grids, tables, or hierarchical data structures**.

Creating a Nested Array

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

console.log(matrix);
// Output: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`for` Loops in JavaScript

A `for` **loop** in JavaScript is used to **execute a block of code multiple times**.

It is commonly used for iterating over arrays, objects, and performing repetitive tasks.

Basic Syntax:

```
for (initialization; condition; iteration) {
    // Code to execute
}
```

Nested `for` Loops in JavaScript

A **nested `for` loop** is a loop inside another loop.

It is commonly used for working with **multi-dimensional arrays, tables, and grids**.

Basic Syntax:

```
for (initialization; condition; iteration) {
    for (initialization; condition; iteration) {
        // Code to execute
    }
}
```

`while` Loops in JavaScript

A **`while` loop** in JavaScript **executes a block of code repeatedlyas long as the condition remains `true`**.

It is useful when the number of iterations is **not known beforehand**.

Basic Syntax:

```
while (condition) {
    // Code to execute
}
```

`break` Keyword in JavaScript

The **`break` keyword** in JavaScript is used to **exit a loop or switch statement prematurely**.

It **immediately stops execution** and exits the loop.

Basic Syntax:

```
for (initialization; condition; iteration) {
    if (exitCondition) {
        break;  // Exit the loop
    }
}
```

Looping Through Arrays in JavaScript

JavaScript provides multiple ways to **iterate over arrays** using different loops.

Each method has its own use case, performance, and readability.

Using a `for` Loop (Traditional Way)

The `for` **loop** gives full control over **iteration** using an index.

```
let fruits = ["Apple", "Banana", "Cherry"];

for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```

/* Output:
Apple
Banana
Cherry
*/

Looping Through Nested Arrays in JavaScript

A **nested array** (or multi-dimensional array) is an array that **contains other arrays as elements**.

To iterate over nested arrays, you need **nested loops**.

Example: Looping Through a 2D Array Using a `for` Loop

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

for (let i = 0; i < matrix.length; i++) {
    for (let j = 0; j < matrix[i].length; j++) {
        console.log(matrix[i][j]);
    }
}
```

/* Output:
1
2
3
4
5
6
7

```
8
9
*/
```

`for...of` Loop in JavaScript

The `for...of` **loop** in JavaScript is used to **iterate over iterable objects** like:

- **Arrays**
- **Strings**
- **Maps**
- **Sets**
- **NodeLists**

Basic Syntax:

```
for (let variable of iterable) {
    // Code to execute
}
```

Nested `for...of` Loop in JavaScript

A **nested** `for...of` **loop** is used to iterate over **multi-dimensional arrays or nested structures**.

It is commonly used when working with **nested arrays (2D arrays), objects inside arrays, or matrices**.

Basic Syntax:

```
for (let outerElement of outerIterable) {
    for (let innerElement of innerIterable) {
        // Code to execute
    }
}
```

Object Literals in JavaScript

An **object literal** in JavaScript is a way to create an object using **key-value pairs**.

It is the most common and convenient method for **defining objects**.

Creating an Object Literal

```
let person = {
    name: "Bhupesh",
    age: 25,
    city: "Waterloo"
};

console.log(person);
// Output: { name: "Bhupesh", age: 25, city: "Waterloo" }
```

Getting Values from an Object in JavaScript

There are multiple ways to **retrieve values** from an object in JavaScript.

1. Using Dot Notation (.)

Dot notation is the simplest way to access object properties.

```
let person = {
    name: "Alice",
    age: 25,
    city: "New York"
};

console.log(person.name);  // Output: "Alice"
console.log(person.age);   // Output: 25
```

2. Using Bracket Notation ([])

```
console.log(person["city"]); // Output: "New York"
```

3. Using Object.values()

```
let car = { brand: "Toyota", model: "Corolla", year: 2023 };

console.log(Object.values(car));
// Output: ["Toyota", "Corolla", 2023]
```

4. Looping Through an Object (for...in)

```
for (let key in person) {
    console.log(person[key]);
}

/* Output:
Alice
25
New York
*/
```

Adding and Updating Values in JavaScript Objects

In JavaScript, you can **add new properties** or **update existing values** in an object using:

- **Dot notation (`.`)**
- **Bracket notation (`[]`)**

1. Adding a New Property

Using Dot Notation:

```
let person = { name: "Alice", age: 25 };

person.city = "New York"; // Adds a new property

console.log(person);
// Output: { name: "Alice", age: 25, city: "New York" }
```

Using Bracket Notation:

```
person["country"] = "USA"; // Adds a new property

console.log(person);
// Output: { name: "Alice", age: 25, city: "New York", country: "USA" }
```

2. Updating an Existing Property

```
person.age = 30; // Updating an existing value
console.log(person.age);
// Output: 30
```

```
person["name"] = "Bob";
console.log(person.name);
// Output: "Bob"
```

3. Adding or Updating Nested Properties

```
let student = {
    name: "John",
    grades: { math: 80, science: 90 }
};

// Updating an existing nested value
student.grades.math = 95;

// Adding a new subject
student.grades.english = 85;

console.log(student);

/* Output:
{
    name: "John",
    grades: { math: 95, science: 90, english: 85 }
}
*/
```

4. Using Object.assign() to Add/Update Multiple Values

```
Object.assign(person, { age: 35, gender: "Female" });

console.log(person);
// Output: { name: "Bob", age: 35, city: "New York", country: "USA", gender:
"Female" }
```

5. Using the Spread Operator (...) to Add/Update

```
let updatedPerson = { ...person, age: 40, hobby: "Reading" };

console.log(updatedPerson);
// Output: { name: "Bob", age: 40, city: "New York", country: "USA", gender:
"Female", hobby: "Reading" }
```

Nested Objects in JavaScript

A **nested object** is an object **inside another object**.

This is useful for **representing hierarchical data** such as **users, products, and configurations**.

Creating a Nested Object

```
let person = {
    name: "Alice",
    age: 25,
    address: {
        street: "123 Main St",
        city: "New York",
        country: "USA"
    }
};

console.log(person);
/* Output:
{
  name: "Alice",
  age: 25,
  address: {
    street: "123 Main St",
    city: "New York",
    country: "USA"
  }
}
*/
```

Array of Objects in JavaScript

An **array of objects** is a collection where **each element is an object**.

This is commonly used for handling **lists of users, products, employees, etc.**

Creating an Array of Objects

```
let users = [
    { name: "Alice", age: 25, city: "New York" },
    { name: "Bob", age: 30, city: "Los Angeles" },
    { name: "Charlie", age: 22, city: "Chicago" }
];

console.log(users);
/* Output:
[
  { name: "Alice", age: 25, city: "New York" },
  { name: "Bob", age: 30, city: "Los Angeles" },
  { name: "Charlie", age: 22, city: "Chicago" }
]
*/
```

Math Object in JavaScript

The **Math object** in JavaScript provides **built-in methods** for performing mathematical operations.

It includes methods for **rounding numbers, generating random values, finding min/max, and more**.

Math Properties (Constants)

Property	Description	Value
`Math.PI`	The value of **π (pi)**	`3.141592653589793`
`Math.E`	Euler's number	`2.718`
`Math.LN2`	Natural logarithm of 2	`0.693`
`Math.LN10`	Natural logarithm of 10	`2.302`

```
console.log(Math.PI);  // 3.141592653589793
console.log(Math.E);   // 2.718281828459045
console.log(Math.round(4.6));  // 5

console.log(Math.floor(4.9));  // 4  Round off to nearest, smallest integer value

console.log(Math.ceil(4.1));   // 5  Round off to nearest, largest integer value

console.log(Math.trunc(4.9));  // 4
console.log(Math.random());
// Output: Random decimal between 0 and 1
console.log(Math.min(5, 3, 9, 1)); // 1
console.log(Math.max(5, 3, 9, 1)); // 9
console.log(Math.pow(2, 3));   // 8
console.log(Math.sqrt(25));    // 5
console.log(Math.cbrt(27));    // 3
console.log(Math.sin(Math.PI / 2)); // 1
console.log(Math.cos(0));           // 1
console.log(Math.tan(Math.PI / 4)); // 1
```

```
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

console.log(getRandomInt(1, 10));
// Output: Random integer between 1 and 10
```

Functions in JavaScript

A **function** in JavaScript is a **block of reusable code** that performs a specific task.

Functions help make code **modular, reusable, and organized**.

Declaring a Function

*1*Function Declaration (Named Function)**

```
function greet() {
    console.log("Hello, world!");
}

greet();
// Output: Hello, world!
```

2. Function Expression (Anonymous Function)

```
const greet = function() {
    console.log("Hello, world!");
};

greet();
// Output: Hello, world!
```

3. Arrow Function (=>) (Shorter Syntax)

```
const greet = () => {
    console.log("Hello, world!");
};

greet();
// Output: Hello, world!
```

Function Parameters and Arguments

4. Function with Parameters

```
function greet(name) {
    console.log(`Hello, ${name}!`);
}

greet("Alice");
// Output: Hello, Alice!
```

5. Function with Default Parameters

```
function greet(name = "Guest") {
    console.log(`Hello, ${name}!`);
}

greet();
// Output: Hello, Guest!
greet("Bob");
// Output: Hello, Bob!
```

Returning Values from Functions

6. Function with Return Statement

```
function add(a, b) {
    return a + b;
}

let sum = add(5, 3);
console.log(sum);
// Output: 8
```

Function Scope

7. Local Scope (Inside Function)

A locally scoped variable is declared inside a function and only accessible within that function.

```
function testScope() {
    let localVar = "I am local";
    console.log(localVar);
}

testScope();
// Output: I am local

console.log(localVar);
// Error: localVar is not defined (only exists inside function)
```

8. Global Scope (Outside Function)

A **globally scoped** variable can be accessed **anywhere in the script**.

```
let globalVar = "I am global";

function testScope() {
    console.log(globalVar);
}

testScope();
// Output: I am global
```

9. Block Scope {} (Using let & const)

Block scope means variables declared inside {} cannot be accessed outside.

```
if (true) {
    let blockVar = "I am block scoped";
    console.log(blockVar); // Accessible inside block
}

console.log(blockVar); // Error: blockVar is not defined
```

10. Scope with var vs let & const

var is NOT block-scoped:

```
if (true) {
    var testVar = "I am using var";
}
console.log(testVar); // ✔️ Accessible outside the block (not safe)
```

let and const are block-scoped:

```
if (true) {
    let testLet = "I am using let";
}
console.log(testLet); // ❌ Error: testLet is not defined
```

11. Lexical Scope (Nested Functions)

A nested function can access variables from its parent function.

```
function outer() {
    let outerVar = "I am outer";

    function inner() {
        console.log(outerVar); // ✔️ Accessible (lexical scope)
    }

    inner();
}

outer();
```

Hoisting in JavaScript

Hoisting is JavaScript's behavior of **moving variable and function declarations to the top** of their scope **before execution**.

This means you can use variables and functions **before declaring them**.

1. Function Hoisting (Works with Function Declarations)

Function **declarations** are **hoisted entirely**, meaning you can call them before defining them.

```
greet(); // ✔️ No Error

function greet() {
    console.log("Hello, world!");
}

/* Output:
Hello, world!
*/
```

Function Expressions in JavaScript

A **function expression** in JavaScript is a way to **define a function inside a variable**.

Unlike **function declarations**, function expressions **are not hoisted** and must be defined before use.

1. Basic Function Expression

```
const greet = function() {
    console.log("Hello, world!");
};

greet();
// Output: Hello, world!
```

2. Named vs. Anonymous Function Expressions

```
const greet = function() {
    console.log("Hello!");
};
```

```
const greet = function greetFunction() {
    console.log("Hello!");
};

greet();
// Output: Hello!

console.log(typeof greetFunction);
// ❌ ReferenceError: greetFunction is not defined (not available outside)
```

3. Function Expressions as Arguments (Callback Functions)

```
setTimeout(function() {
    console.log("Executed after 2 seconds!");
}, 2000);
```

4. Assigning a Function Expression Dynamically

```
let operation;

if (true) {
    operation = function(a, b) {
        return a + b;
    };
} else {
    operation = function(a, b) {
        return a - b;
    };
}

console.log(operation(5, 3));
// Output: 8 (since the first function was assigned)
```

5. Arrow Functions (=>) - A Shorter Alternative

```
const greet = () => {
    console.log("Hello!");
};

greet();
// Output: Hello!
```

Higher-Order Functions in JavaScript

A **higher-order function (HOF)** is a function that **takes another function as an argument** or **returns a function**.

These functions make JavaScript more **powerful, reusable, and modular**.

1. Functions as Arguments

A higher-order function **accepts another function** as a **callback**.

Example: `setTimeout()`

```
setTimeout(function() {
    console.log("Executed after 2 seconds!");
}, 2000);
```

2. Functions Returning Functions

```
function multiplier(factor) {
    return function(number) {
        return number * factor;
    };
}

const double = multiplier(2);
console.log(double(5));  // Output: 10

const triple = multiplier(3);
console.log(triple(5));  // Output: 15
```

Methods in JavaScript

A **method** in JavaScript is a function that is **stored as a property inside an object**.

Methods allow objects to have **behavior**.

1. Creating a Method in an Object

```
let person = {
    name: "Alice",
    age: 25,
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};

person.greet();
// Output: "Hello, my name is Alice"
```

2. Using this in Methods

this refers to the object that calls the method.

```
let car = {
    brand: "Toyota",
    model: "Corolla",
    getDetails: function() {
        return `${this.brand} ${this.model}`;
    }
};

console.log(car.getDetails());
// Output: "Toyota Corolla"
```

3. Using ES6 Method Shorthand

```
let user = {
    name: "John",
    greet() {  // Shorter syntax
        return `Hello, ${this.name}`;
    }
};

console.log(user.greet());
// Output: "Hello, John"
```


4. Adding a Method to an Object Dynamically

```
let person = { name: "Bob" };

person.sayHi = function() {
    return `Hi, I'm ${this.name}!`;
};

console.log(person.sayHi());
// Output: "Hi, I'm Bob!"
```

5. Object Methods: Object.keys(), Object.values(), Object.entries()

```
let student = { name: "Alice", age: 22 };

console.log(Object.keys(student));
// Output: ["name", "age"]
```

```
console.log(Object.values(student));
// Output: ["Alice", 22]
```

```
console.log(Object.entries(student));
/* Output:
[
  ["name", "Alice"],
  ["age", 22]
]
*/
```

6. Method Borrowing (call, apply, bind)

```
let user1 = { name: "Alice" };
let user2 = { name: "Bob" };

function greet() {
    console.log(`Hello, ${this.name}`);
}

greet.call(user1); // Output: "Hello, Alice"
greet.call(user2); // Output: "Hello, Bob"
```

`this` Keyword in JavaScript

The `this` **keyword** in JavaScript refers to **the object that is executing the current function**.

Its value depends on **how and where** the function is called.

1. `this` in the Global Context 🌍

```
console.log(this);
// Output: Window object (in browsers)
// Output: {} (in Node.js)
```

2. this Inside an Object Method

```
let person = {
    name: "Alice",
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};

person.greet();
// Output: "Hello, my name is Alice"
```

3. this in a Regular Function (Undefined in strict mode)

```
function showThis() {
    console.log(this);
}

showThis();
// Output: `window` (in non-strict mode)
// Output: `undefined` (in strict mode)
```

4. this Inside an Arrow Function

```
let user = {
    name: "Bob",
    greet: () => {
        console.log(`Hello, ${this.name}`);
    }
};

user.greet();
// Output: "Hello, undefined"
```

5. this in an Event Listener

In event listeners, this refers to the HTML element that triggered the event.

```
let button = document.createElement("button");
button.innerText = "Click Me";

button.addEventListener("click", function() {
    console.log(this);  // Refers to the button element
});

document.body.appendChild(button);
```

6. this in a Constructor Function

```
function Person(name) {
    this.name = name;
}

let person1 = new Person("Charlie");

console.log(person1.name);
// Output: "Charlie"
```

7. this in Classes (ES6)

```
class Car {
    constructor(brand) {
        this.brand = brand;
    }

    showBrand() {
        console.log(`This car is a ${this.brand}`);
    }
}

let myCar = new Car("Toyota");
myCar.showBrand();
// Output: "This car is a Toyota"
```

8. Controlling this with call(), apply(), and bind()

```
function greet() {
    console.log(`Hello, ${this.name}`);
}

let user1 = { name: "Alice" };
greet.call(user1);
// Output: "Hello, Alice"
```

```
greet.apply(user1);
// Output: "Hello, Alice"
```

```
let newGreet = greet.bind(user1);
newGreet();
// Output: "Hello, Alice"
```

`try...catch` in JavaScript

The `try...catch` **statement** in JavaScript is used for **handling errors** gracefully.

It allows code to **continue executing** even if an error occurs.

1. Basic Syntax

```
try {
    // Code that may throw an error
} catch (error) {
    // Code to handle the error
}
```

2. Using finally (Always Executes)

The finally block runs regardless of whether an error occurs or not.

```
try {
    let num = 10;
    console.log(num.toUpperCase()); // ❌ TypeError
} catch (error) {
    console.log("Error caught:", error.message);
} finally {
    console.log("This runs no matter what!");
}

/* Output:
Error caught: num.toUpperCase is not a function
This runs no matter what!
*/
```

3. Throwing Custom Errors (throw)

You can manually throw errors using the throw keyword.

```
function checkAge(age) {
    if (age < 18) {
        throw new Error("You must be at least 18 years old.");
    }
    return "Access granted.";
}

try {
    console.log(checkAge(16)); // ❌ Throws an error
} catch (error) {
    console.log("Error:", error.message);
}

/* Output:
Error: You must be at least 18 years old.
*/
```

4. Nested try...catch

```
try {
    try {
        throw new Error("Something went wrong!");
    } catch (innerError) {
        console.log("Inner catch:", innerError.message);
        throw new Error("Outer error triggered.");
    }
} catch (outerError) {
    console.log("Outer catch:", outerError.message);
}

/* Output:
Inner catch: Something went wrong!
Outer catch: Outer error triggered.
*/
```

Arrow Functions in JavaScript (=>)

An **arrow function** (=>) is a **shorter and cleaner way** to write functions in JavaScript.

It simplifies function expressions and has **no `this` binding**.

Basic Arrow Function Syntax

```
const greet = () => {
    console.log("Hello, world!");
};

greet();
// Output: "Hello, world!"
```

Arrow Function with Parameters

```
const greet = (name) => {
    return `Hello, ${name}!`;
};

console.log(greet("Alice"));
// Output: "Hello, Alice!"
```

Implicit Return (Shorter Syntax)

When there is only one expression, the return keyword can be omitted.

```
const add = (a, b) => a + b;

console.log(add(5, 3));
// Output: 8
```

Arrow Function Without Parentheses (One Parameter)

```
const square = x => x * x;

console.log(square(4));
// Output: 16
```

Arrow Function in forEach()

```
let numbers = [1, 2, 3];

numbers.forEach(num => console.log(num * 2));

/* Output:
2
4
6
*/
```

Arrow Functions and this (Does NOT Work Like Regular Functions)

Arrow functions do not have their own this.
They inherit this from the surrounding scope.

```
let person = {
    name: "Alice",
    greet: function() {
        setTimeout(() => {
            console.log(`Hello, my name is ${this.name}`);
        }, 1000);
    }
};

person.greet();
// Output (after 1 second): "Hello, my name is Alice"
```

Arrow Functions Should NOT Be Used As Methods

```
let person = {
    name: "Alice",
    greet: () => {
        console.log(`Hello, my name is ${this.name}`);
    }
};

person.greet();
// Output: "Hello, my name is undefined"
```

`setTimeout()` in JavaScript

The `setTimeout()` **function** in JavaScript is used to **execute a function after a specified delay**.

Basic Syntax

```
setTimeout(function, delay);
```

Example: Delayed Execution

```
setTimeout(() => {
    console.log("This message appears after 2 seconds!");
}, 2000);

/* Output (after 2 seconds):
This message appears after 2 seconds!
*/
```

`setInterval()` in JavaScript

The `setInterval()` **function** in JavaScript is used to **execute a function repeatedly** at a specified interval.

Basic Syntax

```
setInterval(function, interval);
```

Example: Repeating a Function Every 2 Seconds

```
setInterval(() => {
    console.log("This message repeats every 2 seconds!");
}, 2000);

/* Output (every 2 seconds):
This message repeats every 2 seconds!
*/
```

Stopping setInterval() with clearInterval()

To stop the interval, use clearInterval() with the interval ID returned by setInterval().

```
let count = 0;
let intervalId = setInterval(() => {
    count++;
    console.log(`Count: ${count}`);

    if (count === 5) {
        clearInterval(intervalId); // Stops the interval after 5 repetitions
        console.log("Interval stopped!");
    }
}, 1000);

/* Output:
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Interval stopped!
*/
```

`this` with Arrow Functions in JavaScript

Arrow functions (`=>`) **do not have their own** `this`.

Instead, they **inherit** `this` **from the surrounding (lexical) scope**.

1. `this` in Regular Functions vs Arrow Functions

```
let person = {
    name: "Alice",
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};

person.greet();
// Output: "Hello, my name is Alice"
```

Issue with this in Regular Functions Inside setTimeout()

```
let person2 = {
    name: "Bob",
    greet: function() {
        setTimeout(function() {
            console.log(`Hello, my name is ${this.name}`);
        }, 1000);
    }
};

person2.greet();
// Output: "Hello, my name is undefined"
```

The function inside setTimeout() creates its own this, which refers to window (global object) instead of person2.

2. Fix Using Arrow Function (this is Inherited)

```
let personFixed = {
    name: "Bob",
    greet: function() {
        setTimeout(() => {
            console.log(`Hello, my name is ${this.name}`);
        }, 1000);
    }
};

personFixed.greet();
// Output (after 1 second): "Hello, my name is Bob"
```

Arrow functions inherit this from the surrounding function (greet), so this.name correctly refers to "Bob".

3. Arrow Functions Inside Object Methods

```
let person3 = {
    name: "Charlie",
    greet: () => {
        console.log(`Hello, my name is ${this.name}`);
    }
};

person3.greet();
// Output: "Hello, my name is undefined"
```

Since arrow functions do not have their own this, this refers to the global object (window in browsers, {} in Node.js).

4. Arrow Functions in Event Listeners (this Issue)

```
document.querySelector("button").addEventListener("click", () => {
    console.log(this);  // `this` refers to `window`, not the button element
});
```

`forEach()` Method in JavaScript

The `forEach()` **method** is used to **iterate over elements in an array** and execute a function for each element.

It is a **cleaner alternative** to `for` loops.

1. Basic Syntax

```
array.forEach(function(element, index, array) {
    // Code to execute
});
```

`map()` Method in JavaScript

The `map()` **method** creates a **new array** by applying a function to each element in an existing array.

It is commonly used for **transforming** arrays.

1. Basic Syntax

```
array.map(function(element, index, array) {
    return transformedElement;
});
```

`map()` Method in JavaScript

The `map()` **method** creates a **new array** by applying a function to each element in an existing array.

It is commonly used for **transforming** arrays.

1. Basic Syntax

```
array.map(function(element, index, array) {
    return transformedElement;
});
```

`every()` Method in JavaScript

The `every()` **method** checks **whether all elements** in an array **pass a given condition**.

It returns `true` **if all elements pass the test, otherwise** `false`.

1. Basic Syntax

```
array.every(function(element, index, array) {
    return condition;
});
```

`some()` Method in JavaScript

The `some()` **method** checks if **at least one element** in an array **passes a given condition**.

It returns `true` **if at least one element satisfies the condition, otherwise** `false`.

1. Basic Syntax

```
array.some(function(element, index, array) {
    return condition;
});
```

`reduce()` Method in JavaScript

The `reduce()` **method** in JavaScript reduces an array to **a single value** by applying a function to each element.

1. Basic Syntax

```
array.reduce(function(accumulator, element, index, array) {
    return updatedAccumulator;
}, initialValue);
```

2. Finding the Maximum Value in an Array

```
let numbers = [4, 8, 2, 10, 5];

let maxNumber = numbers.reduce((max, num) => num > max ? num : max, numbers[0]);

console.log(maxNumber);
// Output: 10
```

Default Parameters in JavaScript

In JavaScript, **default parameters** allow you to **set a default value** for a function parameter.

If the argument is **not provided**, the default value is used.

1. Basic Syntax

```
function functionName(param = defaultValue) {
    // Function body
}
```

Spread Operator (. . .) in JavaScript

The **spread operator (. . .)** in JavaScript is used to **expand arrays, objects, or function arguments**.

It provides a **clean and concise** way to handle data.

1. Basic Syntax

```
const newArray = [...existingArray];
const newObject = { ...existingObject };
```

2. Using Spread with Arrays

```
let fruits = ["Apple", "Banana", "Cherry"];
let newFruits = [...fruits];

console.log(newFruits);
// Output: ["Apple", "Banana", "Cherry"]
```

3. Using Spread with Objects

```
let user = { name: "Alice", age: 25 };
let newUser = { ...user };

console.log(newUser);
// Output: { name: "Alice", age: 25 }
```

4. Using Spread in Function Arguments

```
function sum(a, b, c) {
    return a + b + c;
}

let numbers = [1, 2, 3];

console.log(sum(...numbers));
// Output: 6
```

5. Removing Elements from an Object

```
let user = { name: "Alice", age: 25, city: "New York" };

let { city, ...userWithoutCity } = user;

console.log(userWithoutCity);
// Output: { name: "Alice", age: 25 }
```

Rest Parameter (...) in JavaScript

The **rest parameter (...)** in JavaScript allows functions to accept an **indefinite number of arguments** as an array.

It helps in handling **multiple arguments dynamically**.

1. Basic Syntax

```
function functionName(...restParameter) {
    // Function body
}
```

2. Using Rest Parameters in Functions

```
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4, 5));
// Output: 15
```

3. Using Rest Parameters in Array Destructuring

```
let [first, second, ...rest] = [10, 20, 30, 40, 50];

console.log(first);  // Output: 10
console.log(second); // Output: 20
console.log(rest);   // Output: [30, 40, 50]
```

4. Using Rest Parameters in Object Destructuring

```
let user = { name: "Alice", age: 25, city: "New York", job: "Developer" };

let { name, ...restInfo } = user;

console.log(name);      // Output: "Alice"
console.log(restInfo);  // Output: { age: 25, city: "New York", job: "Developer" }
```

Destructuring in JavaScript

Destructuring in JavaScript allows you to **extract values from arrays and objects** into variables easily.

It makes code **cleaner, shorter, and more readable**.

Array Destructuring

```
let colors = ["Red", "Green", "Blue"];

let [first, second, third] = colors;

console.log(first);  // Output: "Red"
console.log(second); // Output: "Green"
console.log(third);  // Output: "Blue"
```

2. Skipping Elements in Array Destructuring

```
let numbers = [1, 2, 3, 4, 5];

let [first, , third] = numbers;

console.log(first);  // Output: 1
console.log(third);  // Output: 3
```

3. Using Rest (...) in Array Destructuring

```
let fruits = ["Apple", "Banana", "Cherry", "Mango"];

let [first, ...rest] = fruits;

console.log(first);  // Output: "Apple"
console.log(rest);   // Output: ["Banana", "Cherry", "Mango"]
```

4. Object Destructuring

```
let person = { name: "Alice", age: 25, city: "New York" };

let { name, age, city } = person;

console.log(name);  // Output: "Alice"
console.log(age);   // Output: 25
console.log(city);  // Output: "New York"
```

5. Changing Variable Names in Object Destructuring

```
let user = { username: "Bob", age: 30 };

let { username: name, age: userAge } = user;

console.log(name);    // Output: "Bob"
console.log(userAge); // Output: 30
```